

SUBMICRON SYSTEMS ARCHITECTURE PROJECT

Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

Semiannual Technical Report

Caltech Computer Science Technical Report

[5258:TR:87]

11 November 1987

The research described in this report was sponsored by the Defense Advanced Research Projects Agency, DARPA Order numbers 3771 & 6202, and monitored by the Office of Naval Research under contract numbers N00014-79-C-0597 & N00014-87-K-0745.

SUBMICRON SYSTEMS ARCHITECTURE

Semiannual Technical Report

*Department of Computer Science
California Institute of Technology*

[5258:TR:87]

11 November 1987

Reporting Period: 1 April 1987 – 31 October 1987

Principal Investigator: Charles L Seitz

Faculty Investigators: William C. Athas
K. Mani Chandy
Alain J. Martin
Martin Rem
Charles L. Seitz

Sponsored by the
Defense Advanced Research Projects Agency
ARPA Order Numbers 3771 & 6202

Monitored by the
Office of Naval Research
Contract Numbers N00014-79-0597 & N00014-87-K-0745

SUBMICRON SYSTEMS ARCHITECTURE

*Department of Computer Science
California Institute of Technology*

1. Overview and Summary

1.1 Scope of this Report

This document is a summary of the research activities and results for the seven-month period, 1 April 1987 to 31 October 1987, under the Defense Advanced Research Project Agency (DARPA) Submicron Systems Architecture Project. Technical reports covering parts of the project in detail are listed following these summaries, and can be ordered from the Caltech Computer Science Library.

1.2 Objectives

The central theme of this research is the architecture and design of VLSI systems appropriate to a microcircuit technology scaled to submicron feature sizes. Our work is focused on VLSI architecture experiments that involve the design, construction, programming, and use of experimental message-passing concurrent computers, and includes related efforts in concurrent computation and VLSI design.

Additional background information can be found in previous semiannual technical reports [5052:TR:82], [5078:TR:83], [5103:TR:83], [5122:TR:84], [5160:TR:84], [5178:TR:85], [5202:TR:85], [5220:TR:86], [5235:TR:86], [5240:TR:87].

1.3 Highlights

Some highlights of the previous seven months are:

- The Reactive Kernel and version 7 Cosmic Environment, the essential system software for the second-generation Cosmic Cubes, are complete and in daily use on first-generation "cubes" and several other concurrent architectures (sections 2.1, 2.3, 3.2, 3.3, 3.5).
- Concise, a concurrent circuit simulation program, runs on the Intel iPSC/1 and exhibits excellent performance and speedup (section 3.5).
- Several new results in self-timed VLSI designs (sections 4.3-4.6).
- Self-timed routing chips for second-generation multicomputers have been designed, simulated, and sent to fabrication (section 4.7).

2. Architecture Experiments

2.1 Cosmic Cube Project

Bill Athas, Michael Lichter, Wen-King Su, Jakov Seizovic, Chuck Seitz

This section summarizes the current usage and the hardware and software status of the Cosmic Cubes and Intel iPSC/1 d7. Research developments are described in sections 2.3, 3.2, and 3.3 of this report.

The Cosmic Cubes and our Intel iPSC/1 d7 continue to operate reliably. Overall usage has been moderately heavy, and has included both application experiments and major system changes.

One of the largest and most interesting applications from outside of our own group has been a series of Monte Carlo particle-in-cell supersonic flow computations performed by David Goldstein and Professor Brad Sturtevant in Aeronautics at Caltech. In contrast to the usual intensive use of floating point on computational fluid dynamics, this concurrent program employs integer methods. Another interesting application has been a series of neural network learning simulations performed by Alan Blanchard and Professor Yaser Abu-Mostafa. The concurrent circuit simulator, Concise, is now fully operational on the Intel iPSC/1 cubes.

The Cosmic Cubes and Intel iPSC are now operating with reactive programming systems that mimic the programming environment developed for the second generation multicomputers (see section 2.3). The Reactive Kernel has replaced the Cosmic Kernel as the node operating system for the Cosmic Cubes. The Reactive Kernel implements wormhole routing on the Cosmic Cube's program-controlled channels, and supports the same reactive message primitives as the second-generation multicomputers. Programs that use the Cosmic Kernel primitives continue to be supported under a compatibility library. The version 7.2 Cosmic Environment host runtime system has had numerous features added over the past several months for handling mixed types of multicomputer hardware and message protocols.

Neither the 64-node nor 8-node Cosmic Cubes has exhibited a hard failure in this seven-month period. The cubes have now logged about 2.9 million node-hours with only three hard failures. The calculated node MTBF of 100,000 hours reported before these machines were constructed was extremely conservative. A node MTBF in excess of 1,000,000 hours is probable, and can now be stated at a 50% confidence level.

Our Intel iPSC/1 d7 (128 nodes) was contributed to the Submicron Systems Architecture Project as a part of the license agreement between the Caltech and Intel, and is accessible via the ARPAnet to other DARPA researchers who may wish to experiment with it. To request an account, please contact chuck@vlsi.caltech.edu. Delivery of an Intel iPSC/2 and a new Ametek system is anticipated within the next six months.

2.2 Mosaic Project

Bill Athas, Charles Flaig, Glenn Lewis, Don Speck, Wen-King Su, Chuck Seitz

The Mosaic C is a message-passing MIMD multicomputer with single-chip nodes. The stipulation that the nodes fit on a single chip limits the storage for each node, so that relatively fine-grain concurrent programming techniques must be used. We are working toward building a 16K-node Mosaic system using nodes fabricated in $1.2\mu\text{m}$ CMOS technology, with a near-term milestone of a 1K-node system using nodes fabricated in $2\mu\text{m}$ CMOS.

The status of the Mosaic C chip design is described in section 4.1, and the current work on the Cantor programming system that we shall use for programming the Mosaic is described in section 3.1.

2.3 Second-Generation Cosmic Cubes*

Chuck Seitz, Alain Martin, Bill Athas, Charles Flaig, Jakov Seizovic, Craig Steele, Wen-King Su

In September 1986, we started simultaneous joint projects with two companies, Intel Scientific Computers and Ametek Computer Research Division, to build prototype second-generation message-passing multicomputer systems. The companies are providing all necessary parts, assembly, and logistical support for constructing the prototypes to our specifications and designs, and are working with us in developing the system software. Intel and Ametek have non-exclusive licenses to patents on the Cosmic Cube architecture and message-passing mechanisms, several later patents, and ongoing work at Caltech. They also have non-exclusive resale licenses for the system software developed in our group. Both companies are manufacturers of first-generation message-passing multicomputers based on the Cosmic Cube, and are licensed to produce the second-generation machines. They will contribute systems to the project; these will be made accessible to researchers via the ARPAnet, as is currently the case with our Cosmic Cubes and iPSC/1.

The general hardware characteristics of these second-generation multicomputers are that the node processors are about ten times faster than the first-generation machines, but the message performance for non-localized message traffic is about 1000 times faster. The message network uses a low-dimension graph, rather than the binary n -cube, and wormhole routing. The machines can be scaled over a wide range of N , but a 256-node system is the "centerline" design point. We are already well underway with these projects. However, because of proprietary concerns, the

* This segment of our research is sponsored jointly by DARPA and by grants from Intel Scientific Computers (Beaverton, Oregon) and Ametek Computer Research Division (Arcadia, California).

schedules and certain details of the designs are not included in this report, but have been reported separately to the DARPA management.

In September 1987, Intel announced the iPSC/2. This system is an iPSC/1 node upgrade with these major improvements:

- (1) The 80286 node processor has been replaced with a 80386 processor, which is nearly ten times faster.
- (2) Node memory capacity has been increased to 8 MB.
- (3) The message-passing communication system, while still based on a binary n -cube, employs an Intel implementation of the wormhole routing techniques that were developed and prototyped in our group. Both the message latency and the sensitivity of the latency to locality in the binary n -cube have been substantially reduced.

The cube host has also been upgraded to an Intel 301, which runs system V Unix and has socket and NFS implementations.

The iPSC/2 is a system that is, in comparison with our targets for second-generation multicomputers, intermediate between the first and second generations. It allows iPSC/1 systems to be upgraded, and is upwardly compatible in application software. It is our intention to port the RK to the iPSC/2 and the Cosmic Environment to the 301 host as soon as we have hardware on site.

3. Concurrent Computation

3.1 Cantor

Bill Athas, Nanette Jackson, Chuck Seitz

The development of the Cantor programming system is motivated by the need to express concurrent programs in a way that is suitable for execution on fine-grain multicomputers. In particular, we are developing object-oriented programming techniques that are suited to the Mosaic C multicomputer, with its strict limitations on the size of objects within a computation. Naturally, the techniques developed to program effectively in this fine-grain object environment are compatible with systems permitting computation at a coarser granularity.

A stable version (2.0) of the Cantor programming system has been in service at Caltech and at other universities and research institutions for the past eight months. Our research activities have included both the continued development of application programs and of the Cantor system itself.

In the area of application programs, we have expanded the process of evaluating the expressive capability of the language by designing and coding fine-grain solutions to some standard programming problems. In addition to the set of benchmark programs that have been reported previously, we now have several implementations of Cantor programs that solve the all-points-shortest-path and maxflow problems, and have even written a rudimentary chess program. The programs that have already been developed have yielded insight about the techniques necessary to satisfy the special programming requirements of fine-grain multicomputers and about the utility of Cantor to express these techniques. Analysis of these programs has also suggested improvements to Cantor.

From our evaluation of the Cantor benchmark programs and from our experiences with the Cantor Engine (see section 4.2), several refinements have been considered and implemented in an experimental Cantor version 2.1. Runtime type checking has been replaced by compile-time type checking. The syntax of Cantor has been augmented with type declarations and an additional pass has been added to the compiler front end to check that all types are compatible. The decision in favor of static type checking was based upon three observations: Dynamic typing is rarely used in the benchmark programs; the compile-time inferencing of variable types is largely unsuccessful because message variables are late-binding; and the silicon area dedicated in the tagpath of the Cantor Engine chip for processing type information is quite large.

By switching to static type checking, the storage class for every object variable is known at compile time. To experiment with this new capability, a notation and the semantics for vectors have been included in version 2.1. The built-in arithmetic

operators work over both scalars and vectors. The benchmark programs have been recoded to take advantage of the vector notation. For version 2.1, the size of every vector must be known at compile time. From our preliminary evaluation of programs written in Cantor 2.1, this requirement is often a severe restriction in program expression. For Cantor 2.2, we plan to have *open* vectors, which are akin to the open arrays of Modula-II.

The semantics of Cantor objects have been liberalized to allow objects to be created in a running state and to run indefinitely without consuming any messages. Programs may therefore be written which violate the *reactive property*. In the future we plan to provide a program flow analysis tool that will check Cantor programs for possible violations of the reactive property.

Over the next few months, we plan also to continue the development of our Cantor program library and the accompanying enhancements of Cantor, culminating in some larger programming examples and perhaps some generalizations about techniques for programming fine-grain machines.

3.2 Reactive Primitives for the Cosmic Environment

Wen-King Su, Chuck Seitz

The Cosmic Environment, our generic, portable multicomputer interface, has been revised extensively, and is now being distributed in version 7.2.

We have switched over (relatively painlessly, considering the number of users involved) to a new set of reactive message primitives:

`m = xmalloc(1)` Allocate a message buffer.

`xfree(m)` Deallocate a message buffer.

`xsend(m,n,p)` Same as `xfree`, with the side-effect of sending a copy of the buffer, `m`, to process `(n,p)`.

`m = xrecvb()` Same as `xmalloc`, with the side effect of filling the buffer with the next message to arrive. The buffer length is set to the length of the message.

`m = xrecv()` A non-blocking form of `xrecvb`, which will return NULL if no message has arrived.

`l = xlength(m)` Return the length of the buffer.

Implementing other higher-level message primitives in terms of this set has proved to be straightforward and very efficient. The Cosmic Environment and the various multicomputers it supports now implement the programming environment we have developed for second-generation multicomputers.

In addition to supporting the original Cosmic Cube and the iPSC/1, the version 7.2 Cosmic Environment supports a new cube type in which a group of NFS-connected workstations are treated as a multicomputer. This new cube type, called a *ghost cube*, is operationally identical to the other cubes that we support.

In anticipation of more specialized and sophisticated user interfaces, the Cosmic Environment makes the support of multiple message protocols quite simple. Each protocol/hardware combination now comes with its own library and server processes. For example, the older `sendq` and `recvq` primitives are available via the “cosmic” protocol. The relationship between the protocols and the Cosmic Environment is not unlike that of a shell to a Unix system.

The Cosmic Environment continues to be ported to other Unix systems. The developers of Concise have ported the Cosmic Environment to a Sequent (see section 3.5), and other ports to several high-speed multiprocessors are in progress.

3.3 The Reactive Kernel

Jakov Seizovic, Wen-King Su, William C. Athas, Chuck Seitz

The Reactive Kernel is the new node operating system for the second-generation Cosmic Cubes. Implemented initially on the original Cosmic Cubes, it has been running successfully for users during the last month. One of the issues that was recognized as being very important from the early stage of the design is portability. The Reactive Kernel is written in C, and is being ported very successfully to one of the second-generation multicomputers while retaining about 90% of the original code.

The Reactive Kernel uses the new set of communication primitives that are described in section 3.2. These represent our choice of a minimal set of primitives, and form the base on which any other set of primitives can be built.

The Reactive Kernel is structured in two main layers: the inner kernel and a set of *handlers*. A handler is similar to a kernel process in other operating systems, but is restricted to satisfy the reactive property.

The inner kernel contains routines that implement the reactive communication primitives, a dispatch loop, and a message interface. The inner kernel has no knowledge of user processes. It interacts only with handlers. The central part of the inner kernel is the dispatch loop that fetches messages from the receive queue and delivers them to the handlers, according to the *tag* that is given to each message as it is sent. Different tags may require that the message be discarded, turned into the code segment, reassembled, or further delivered (for example, to a user process).

The idea of handlers was inspired by the Actor semantics, by the behavior of Cantor objects, and by the operation of distributed event-driven simulators. A handler is invoked when there is a message tagged for it; as a reaction to the message,

it may *send* new messages and create *new* handlers. A handler may run only for a bounded period of time, after which it is required to specify its replacement behavior; *ie*, it can *become* a new handler, self-destruct, or replace itself by the identical handler (default action). Unlike the usual kernel process, the state that is saved between invocations of the same handler is retained only in storage that has been explicitly allocated by the handler.

A generic reactive handler has been implemented. Its main job is to provide the support for the standard user processes. Control is given to the user process when there is a message for it, and the user returns control back at the *choice points*, marked with the execution of `xrecv()`. Since context switches generally occur at these well-defined places in user processes, their cost has been reduced to that of the system function call.

The absence of the type mechanism in the reactive primitives means that the user process cannot exercise discretion; *ie*, it has to accept the messages in the order that they arrive. However, discretion mechanisms can be built on top of reactive primitives. An example is the library of the Cosmic-style primitives (`sendq()`, `recvq()`) that have been written in terms of reactive primitives. This library enables any user program written for the first-generation Cosmic Cubes to run under the Reactive Kernel.

3.4 Object-Oriented Event-Driven Simulation

Wen-King Su, Chuck Seitz

Object-oriented simulations map well onto message-passing multicomputers. Elements in a simulation are represented by processes and the information flow between them is coded in the messages.

A drawback of this simplistic mapping is that in order to mimic the continuous flow of information that exists in a real system, each element has to keep other elements constantly up to date on its own state. The multicomputer on which such a simulator runs would be overwhelmed by the volume of message traffic it produces.

A solution to this problem is for each element to lump the information produced over a period of time, and to send an interval as one message instead of as a continuous stream of messages. Different deterministic simulators differ by the way the sizes of the intervals are chosen and by the way these intervals of information are conveyed to their proper destinations. An "indefinite-lazy" simulator is loosely defined as one whose elements accumulate indefinitely large intervals before sending them off.

From the standpoint of containing the volume of message traffic, the lazier a simulator can run, the better. However, excessive laziness can introduce deadlock. For example, consider the situation in which a set of elements in a ring are all

waiting to fill up their own interval quotas, and none is willing to send any part of its own interval down the line. Other research on how to avoid such a situation includes that of Chandy and Misra, who use timely generation of null messages and runtime deadlock detection.

The method we use is based on the indefinite-lazy simulation model. Under this model, as has been reported previously, the elements are allowed to do whatever they like so long as there are messages in the input queue of the node. When the receive queue runs dry, the elements are selectively forced to emit the intervals they may have accumulated. Such simulators can be shown to be free of deadlock. Furthermore, on top of the indefinite-lazy simulation model, we can put any type of laziness factor into the elements themselves without causing deadlock. We are thus able to study simulation heuristics unhindered by the question of deadlock.

Based on the indefinite-lazy simulation model, a number of logic circuit simulators with different overall laziness attributes have been written to investigate the performance associated with various heuristics. The best of these simulators exhibit a linear speedup with the size of the machine up to about ten logic elements per node. These results are preliminary, and there are still more heuristics to incorporate into the simulators, as well as more types of logic circuits to simulate.

3.5 Concise — A Concurrent Circuit Simulator*

Sven Mattisson, Lena Peterson, Chuck Seitz

The concurrent circuit simulation program, Concise, is now run routinely on the Intel iPSC/1 under the Cosmic Environment. Initial timing results indicate that the previously reported speedups based on simulations are realized in practice. The speedup obtained by Concise is limited in most circuits by load imbalance rather than by message latency. Speedups better than $N/2$ are obtained when the number of electrical circuit nodes is about three times larger than the number of computing nodes, even with circuits of more than 100 electrical nodes. The speedup approaches N when the number of electrical circuit nodes is considerably larger than the number of computing nodes.

Concise has proved to be very stable due to the dynamic window splitting, even on circuits with tight feedback loops. We have not yet seen the “internal time step too small” message that is so well known to users of SPICE.

Concise currently runs under the Cosmic Environment with the reactive primitives on Unix computers; on all forms of multicomputers, including ghost cubes; and also on a Sequent, a shared memory multiprocessor. The operating system on

* This segment of our research is a joint project between the Caltech Submicron Systems Architecture Project and the Department of Applied Electronics at the University of Lund, Sweden.

the Sequent permits good CPU utilization when the number of simulation processes exceeds the number of available CPUs. Thus, even though intended for multicomputers with up to hundreds of nodes, Concise has good performance on a 10-node multiprocessor. We regard this as a good demonstration of the portability and convenience of the Cosmic Environment, in that it allows this complex program to run unchanged on a variety of concurrent and sequential computer architectures.

Presently we are working on a partitioning algorithm that will enable Concise to balance its load and to improve its convergence rate. The present partitioning scheme is static, but a dynamic scheme will be attempted in the near future.

3.6 Preliminary Design of a "Page Kernel"

Craig S. Steele, Chuck Seitz

Second-generation multicomputers are distinguished from their predecessors primarily by dramatically improved communications subsystems; but their node microprocessors are substantially more capable as well, and the memory-management hardware support, in particular, is much more sophisticated.

Preliminary work is underway to design a "page kernel," which is an alternate programming interface to the message system. A page kernel is implemented using memory access protection mechanisms. Current memory management hardware permits a process to address many different pages (or segments), each of which may have individual access rights. For example, while one page might be readable, an attempt to change its contents by writing to it could cause a hardware exception; another page used for the process stack would not have access restrictions. The new generation of processors all have large address spaces containing many pages, each of which can be treated as all or part of a distinct data object.

A programming model based on hardware-controlled memory access is an interesting variation of the message-passing model. Like the reactive model, message receipt becomes less obvious than with the Cosmic Kernel model. When a process attempts to read a protected datum, the kernel can be triggered implicitly to place valid data in that page by the exception-handling routine. This action can be the equivalent of an explicit receive. Likewise, an attempt to write to a protected page can replace an explicit send function.

The availability of such memory-protection hardware would be novel for multicomputers but is old news for conventional mid-range sequential machines such as VAXen. Why should this notion be interesting for multicomputers when it has not been exploited for traditional systems?

There are several reasons why the page model has not been used. First, process context switch times, even for exception handlers, are large. Second, page protection is not in itself an obviously adequate set of synchronization primitives. Third, user modification of hardware registers is unwelcome in multiuser environments, even if

the operating system provides the option. This last problem is largely irrelevant for multicomputers, which are generally space-shared rather than time-shared.

There are two programming environments where the other difficulties are believed to be tractable. One is the reactive model; the other the chaotic model.

The processes of the reactive model explicitly tell the kernel the choice points at which it may consider rescheduling them by performing a context switch. This trick reduces the amount of process state that must be saved, and improves context switch speed. In addition, the explicit specification of a choice point implies that the action is complete and the output valid. This provides the information lacking in the naive use of page exceptions. Consider an inner product process which accepts two vectors as input and produces one as output. While one can easily mark the output datum as “dirty” at the initial write, when should that information be sent? The decision of when the output is logically consistent must be made by the programmer, for example, by allowing process termination.

Recasting the reactive process model in this form changes the primary programming effort from the imperative style of specifying actions to the declarative style of specifying logical interconnections. It is not obviously better or worse, but different. The page scheme is probably a superior implementation for the second class of applications, the iterative chaotic methods that model inherently convergent problems. Naturally, many physical simulations fall into this class, but many traditional computer science tasks can be put in such a form as well. If the calculation is well-behaved, synchronization of input data can be less important and knowing the exact arrival time of each datum may be unnecessary — using the latest data may be all that is desired. The page model is well-suited to asynchronously updating the process state in relaxation calculations.

The process address space is, in effect, distributed across the machine. Practical considerations prevent this address space from being truly global; there aren't enough pages to let every datum have the same name across a large machine. However, the high communications performance of the second-generation multicomputers allows many of the functions of a shared-memory, global-name-space multiprocessor to be emulated efficiently.

4. VLSI Design

4.1 Mosaic Elements

Bill Athas, Charles Flaig, Glenn Lewis, Don Speck, Wen-King Su, Chuck Seitz

The Mosaic C chip is composed of three main parts: RAM & ROM, channels, and processor. Our strategy for verifying the design of this very complex chip and characterizing its yield on MOSIS runs is initially to fabricate and test the three main parts separately. After the parts have been well characterized, their layouts will be combined onto a single chip.

The target technology for the Mosaic C is MOSIS SCMOS with $0.6\mu\text{m} \leq \lambda \leq 1.5\mu\text{m}$. Target maximum chip size is 36mm^2 , or $100\text{M}\lambda^2$ with $\lambda = 0.6\mu\text{m}$, and $16\text{M}\lambda^2$ with $\lambda = 1.5\mu\text{m}$. Speed, storage size, and top-level floorplan will necessarily vary with feature size.

The architecture of the Mosaic C and the design of the Mosaic C chip are described in the last two semiannual technical reports, [5235:TR:86] and [5240:TR:87].

4.1.1 Mosaic C dRAM

Our basic strategy has been to develop a 4-transistor dRAM that is a low-risk design with a relatively large area, and a 2-transistor dRAM that is a higher-risk design but has a relatively small area.

Four-transistor dRAM

Not believing the yield reported in our previous semiannual technical report of only three out of twelve on an 8-kilobit chip fabricated in a $3\mu\text{m}$ SCMOS process, we obtained additional bonded chips for a recount. The revised yield figure is eleven out of 30, which does not change any of the conclusions from the last report.

Timing tests show that the prototype has good operating margins. The chip functions correctly on power supply voltages of 3 to 6 volts, and tolerates 5 to 15 nanoseconds of clock skew (depending on supply voltage). However, the sensing phase takes 70 nanoseconds at 5 volts, about twice as long as planned.

To check the yield and speed, the chip is being refabricated on a $1.6\mu\text{m}$ run. The layout has been updated to conform to changes in the MOSIS design rules.

Two-transistor dRAM

After much searching for structures that would automatically give an appropriate sense current for the sense amplifier, we concluded that we were not going to find one that was tolerant of power supply variations. We decided instead to build a more *ad hoc* circuit having the right power supply dependence, and trust in SPICE to get the currents correct.

The general approach is to apply linear voltage ramps to the gates of successively larger transistors, with the completion of each ramp causing the start of the next. Each ramp rate is set by the size of an n -channel transistor that drives a capacitance via a p -channel current mirror, to maintain a constant current. With appropriate choice of ramp rates and transistor sizes, three stages produce an adequate approximation to an ideal sense current. The first ramp comes from a dummy select line that matches the delay of the polysilicon select lines crossing the cell array.

Since the Mosaic C does not employ error correction, alpha-particle-induced bit errors are a serious concern. The strategy suggested to us is to put the cell array in a p -well, whose depletion region will repel any stray minority carriers that diffuse up from the substrate, where alpha particles generate the most carriers. Unfortunately, the heavy doping necessary to form a well greatly increases the diffusion capacitance, which reduces the bit-line voltage swing and hence the noise immunity. With less voltage to work with, it becomes important to get as much of the cell charge onto the bit lines as practical before sensing.

Polysilicon select lines and single-transistor pass gates may seem to be a slow way to do this, but it happens that as the source and drain voltages of the access transistors rise and turn the transistor off, the gate capacitance bootstraps the select line slightly above the power supply voltage. This means that the select transistors can be wider than one might expect, given the resistance of the polysilicon wire, and this helps squeeze out more charge. Not being able to squeeze out all of the charge is both a curse and a blessing. When an alpha particle strikes, it is the charge that is hard to get that is lost, so the effect on bit-line voltage swing is not as large as it first seems.

The limited voltage swing puts a premium on exactly equal precharge voltages. Simplistic simulations show that this should be easy to attain; however, when the power supply voltage moves continuously during precharge, it becomes more difficult, requiring either large precharge transistors that will glitch the power supply, or a third transistor that will short the bit lines together during precharge.

The limited voltage swing also puts a premium on writing full voltage levels into the cells. Because the write transistors must behave as isolation cascodes during reading, they must be single-transistor pass gates, which can pass a voltage just as high as the cell select transistors can pass, but do it quite slowly. This could be combatted with large transistors (which increase the sneak current through the sense current wire), with a complementary sense amplifier (decreasing the density), or by writing early enough in the cycle that the bit lines have not yet had a chance to fall. It is not yet clear which approach is best, so this is an area meriting further study.

Reading needs additional study as well, because it loads the sense amplifier and worsens the low voltage that can be restored into the cell. The effective cell capacitance is maximum for low voltages, so the low level is important.

4.1.2 Channels

The test layout for Mosaic channels of minimum flit width (as described in the last report) returned from fabrication at the end of April, and was tested by the end of May. This test chip was found to function correctly in all respects except for blocking messages when a succeeding channel was full. Analysis of the involved layout revealed a short circuit in some of the switch configurations used to propagate the acknowledge signal. These shorts did not show up in simulation because of a set of improperly constructed test vectors which conspired with the nature of the defect to give the expected result. It was not possible to test the router at speed since many internal nodes had been tapped to produce debugging outputs.

The layout for the switch was corrected, and was successfully simulated using corrected test vectors. The layout for the one-dimensional router was modified for 5-bit-wide flits (4-bit-wide data), simulated, and again sent out for fabrication. This version is expected to work correctly, so the debugging taps were removed to allow us to make speed tests.

Work is continuing on the processor interface for the channels, and on making a mock-up of a full Mosaic C chip. The "mock-Mosaic" will consist of a 2D router with the same pad frame as the Mosaic C; but, whereas the full-scale Mosaic C directs messages to the node computer, the mock-Mosaic will redirect those messages back into the network. These chips will be useful for testing the top assembly packaging and host interfaces.

4.1.3 Processor

The Mosaic C processor datapath layout has been further refined. The current design simulates correctly with MOSSIM. The size of this 16-bit datapath, including 24 general registers, ALU, shifter, condition flags, seven addressing and address limit registers, and address arithmetic, is approximately $1000\lambda \times 3500\lambda$.

The microcode for Mosaic C is also complete, and has been tested using a detailed instruction-level simulator. Several improvements have been made to the instruction set in this period to make it a better match to the Cantor code generator. The microcode has also been partitioned in several ways to try to discover ways to reduce the number of literals per implicant. The goal of this partitioning is to try to solve by organizational means some problems in keeping the control PLA from being the critical timing path.

4.2 Cantor Engine

Bill Athas, Nanette Jackson, Jakov Seizovic

The Cantor Engine, as described in our previous semiannual technical report, is an experimental VLSI implementation of a message-driven instruction processor

architecture. The design has made substantial progress over the past seven months, with individual sub-systems submitted for fabrication. The layout for the bus interface unit and the datapath have been completed. The bus interface unit was sent to MOSIS for fabrication in mid-June and chips were received at the end of September. The datapath has been assembled and is expected to be sent to MOSIS before the end of November.

4.3 A Self-Timed Circuit for the Lazy Stack

Steve Burns, Alain Martin

The design of a self-timed VLSI circuit for a “lazy” stack has been used to illustrate a new aspect of our method for synthesizing self-timed circuits from communicating processes. In this design, the control part is first constructed according to the standard method, and the data path is then added, using a systematic compilation technique for input and output. This separation of control and data parts not only makes the compilation easier but it also leads — in most cases — to simpler circuits: Because read and write operations on each variable are grouped together in the data path, duplication of identical circuitry is avoided.

We produced the layout using a semi-automated standard-cell place-and-route design tool.

A stack consisting of 24 cells, each capable of holding four bits of data, has been fabricated. All twelve packaged chips returned from MOSIS function correctly. The chips perform well: The average gate-delay is approximately $2ns$, and a worst-case delay of $60ns$ is needed to add a value to the stack. Because of the self-timed nature of the design, the chips function correctly, but much more slowly, at extremely low power-supply voltages (down to 0.7 volts).

4.4 Automatic Compilation of Communicating Processes into Self-Timed VLSI Circuits

Steve Burns, Alain Martin

We have constructed a first compiler which automatically translates multi-process programs into self-timed circuits. The translation is directed by the syntax of the program; *ie*, we derive a standard implementation for each language construct. Hence, the connectivity of the parse tree mirrors the connectivity of the circuit.

Although this first compiler does not use all heuristics of hand compilation, possible inefficiencies of the compiled circuits are almost entirely eliminated by “peep-hole optimizations.” (We are still refining this optimization step.)

The translation algorithm also provides a constructive proof that any program in our notation can be compiled into a self-timed VLSI circuit whose size is proportional to the size of the source code.

The output of the compiler is a network of operators (and, or, C-element, flip-flop, etc.) for which we have designed a library of standard CMOS cells. The output of the compiler may be fed into MOSIS's FUSION service in order to produce layouts from programs completely automatically. The compiler consists of approximately 1,000 PROLOG clauses and translates an average half-page program in less than ten seconds. We are currently comparing the results of the compiler to the results of various hand compilations to determine the quality of the compiled circuits.

4.5 Self-Timed Implementations of Systolic Algorithms

Pieter Hazewindus, Tony Lee, Alain Martin

So far, most programs that we have compiled into self-timed circuits have been programs that either synchronize processes or buffer and route data. We have not investigated in great detail programs that involve arithmetic. (The most significant chip we have built containing arithmetic is the " $3x + 1$ engine.") We have started a systematic study of this type of self-timed circuit, and, as a first step, are analyzing the compilation of systolic algorithms.

The simplest "systolic" interconnection is an array of processes, with each process repeatedly receiving a value from its left and right neighbors, and sending values to its right and left neighbors. Because of the regularity of the communication behavior, it is possible to acknowledge the receipt of a value from the left neighbor by sending a value to this neighbor, and similarly for the right neighbor. Using this acknowledgment protocol, we not only minimize the bandwidth of channels, but also significantly reduce the size of the circuits required to implement systolic algorithms.

Using this method, we have designed an 8-bit self-timed systolic multiplier that is currently being fabricated by MOSIS in a "TinyChip" run (28-pin package). In a systolic multiplier, each process receives a bit of the multiplier from the left neighbor and a partial sum from the right neighbor, then sends the same multiplier to the right and the new computed partial sum to the left. With the above communication protocol, it is possible to have the values sent be any function of the values received. In particular, these values need be neither binary nor dual-rail encoded variables.

We are currently investigating the generalization of this method for alternate architectures, such as a hexagonal grid of processes.

We are also investigating the implementation of signal-processing algorithms into self-timed circuits. Possible applications include clockless encoding/decoding devices; off-line data processing units; and, with the development of proper interfaces, real-time asynchronous signal-processing systems.

We have chosen a bit-serial approach, since it makes efficient use of relatively small amounts of circuitry and requires only bit-wide communication channels — two properties that fit well within a self-timed discipline. The general design strategy being pursued is to develop a library of asynchronous utility modules, such as multipliers and multiplexors, and then to compose them together to form the target system. This method has been applied to the construction of a prototype asynchronous finite impulse response filter.

4.6 Formal Program Transformations for Self-Timed Circuit Synthesis

Alain Martin, David Long

The efficiency and generality of our synthesis method for self-timed VLSI circuits can be attributed in large part to the formal program transformation techniques on which it is based. These techniques make it possible for the designer to modify and refine its design in a certain direction while maintaining its functionality. (Each “legal” transformation guarantees that the resulting program will be semantically equivalent to the source program.)

As a part of a systematic effort to better understand when and how these transformations should be applied, we have written a program that performs some of these transformations automatically. The program (written PROLOG) takes a straightline handshaking expansion and produces a circuit by applying the three main transformations: reshuffling, state-variable introduction, and production rule expansion.

In order to select the “best” solution each time a choice is possible, a metric has been attached to the programs. So far, all solutions produced correspond to the solution produced by the hand compilation of the same program.

4.7 A Self-Timed Mesh Routing Chip

Charles Flaig, Chuck Seitz

We have developed a general scheme for designing routing chips that is based on routing automata, as described in previous reports. We have now developed CMOS designs and layouts, and a stylized way of assembling them, that enables us to assemble whatever kind of routing chip we might like within a day or two.

An asynchronous routing chip targeted for use in second-generation multicomputers was completed, simulated, and sent off for fabrication at the beginning of September. We expect to receive first silicon shortly.

Testing of the layout of this chip was carried out in two stages. The first stage involved the hand application of test vectors to try to exercise each section of each data path. Once all errors had been removed using this method, a program

written by Wen-king Su was used to generate messages and apply simulation test vectors using a high-level model of the behavior of the router and the data formats involved. This program also compares the results of each stage of the simulation with its model, and prints a table of applied and generated messages in a coherent form suitable for easy manual checking. This extensive workout revealed several obscure errors, which were then corrected. We are confident that the logic of the chip is correct. The questions remaining to be answered by testing the fabricated chips involve possible analog timing glitches that are not caught by switch level simulation, and the actual attainable speed.

4.8 Adaptive Routing in Multicomputer Networks

John Y. Ngai, Chuck Seitz

Our investigation continues in exploring the use of adaptive routing strategies to improve and sustain the performance of communication networks in moderate to heavy message traffic conditions. The summary of the various findings of the initial phase of our investigation is described in detail in our technical report [5246:TR:87], which focuses on the examination of issues fundamental to any communication network supporting reliable concurrent computation. The problems of communication deadlock freedom, packet delivery assurance, and packet injection assurance are addressed in the framework of the simple *packet exchange* model.

Based on our investigation of the simple packet exchange model, we realized that the critical assumption underlying our assurance of deadlock freedom in spite of arbitrary routing patterns is our ability to tightly match the incoming and outgoing data rates of a node. This is an assumption which is valid due to the tightly coupled structure of multicomputer communication networks. By combining this assumption and the *preemption* strategy, we are able to generalize our previous results concerning deadlock freedom, packet delivery, and injection assurances to *virtual cut-through** style routing. In this adaptive cut-through switching framework, the forwarding of incoming packets with more than one profitable outgoing channel can start as soon as the first of its profitable outgoing channels becomes available, without having to wait for the arrival of the entire packet. Analyses and simulations have demonstrated that packet latency under cut-through switching is substantially lower than that of the corresponding store-and-forward switching techniques used in light to moderate message loadings.

The adaptive cut-through switching extension provides us with a competitive routing technique that holds promise of delivering performance surpassing that of

* P. Kermani and L. Kleinrock, "Virtual Cut-Through: A New Computer Communication Switching Technique," *Computer Networks* 3(4) pp. 267-286 (Sept. 1979).

the already highly-evolved oblivious wormhole routing technique. Our preliminary simulation results have identified three regions of performance comparisons:

- (1) Light offered message loading, where both the adaptive and oblivious techniques deliver approximately the same delay and throughput;
- (2) Moderately offered message loading, where the achieved throughput remains approximately the same, while the message delay delivered by the adaptive technique is significantly lower; and
- (3) Heavy offered message loading, where the oblivious technique breaks down with message delays and the source queue lengths increase out of bound. In contrast, the adaptive technique continues to deliver acceptable performance with throughput approaching that of the available physical network bandwidth. The message delay, on the other hand, approaches that of the conventional store-and-forward switching.

Our current work is mainly focused on consolidating the adaptive cut-through switching model; a technical report summarizing our new results is under preparation. Also under examination are various local heuristic routing assignment strategies and their comparisons through extensive simulations.

California Institute of Technology
Computer Science Department, 256-80
Pasadena CA 91125

Technical Reports

October, 1987

Available from the Computer Science Department Library

Prices include postage and help to defray our printing and mailing costs.

Publication Order Form

If you wish to order any of the reports listed, complete this form and return it with your check or international money order (in U.S. dollars) payable to CALTECH. Prepayment is required for all materials.

___5256:TR:87	\$2.00	<i>Synthesis Method for Self-timed VLSI Circuits</i> , Martin, Alain current supply only: see <i>Proc. ICCD'87: 1987 IEEE Int'l. Conf. on Computer Design</i> , 224-229, Oct'87
___5252:TR:87	\$2.00	<i>C Programmer's Abbreviated Guide to Multicomputer Programming</i> , Seitz, Charles L.
___5251:TR:87	\$2.00	<i>Conditional Knowledge as a Basis for Distributed Simulation</i> , Chandy, K. Mani and Jay Misra
___5250:TR:87	\$10.00	<i>Images, Numerical Analysis of Singularities and Shock Filters</i> , PhD Thesis Rudin, Leonid Iakov
___5249:TR:87	\$6.00	<i>Logic from Programming Language Semantics</i> , PhD Thesis Choo, Young-il
___5247:TR:87	\$6.00	<i>VLSI Concurrent Computation for Music Synthesis</i> , PhD Thesis Wawrzynek, John
___5246:TR:87	\$3.00	<i>Framework for Adaptive Routing</i> Ngai, John Y and Charles L. Seitz
___5244:TR:87	\$3.00	<i>Multicomputers</i> Athas, William C and Charles L Seitz
___5243:TR:87	\$5.00	<i>Resource-Bounded Category and Measure in Exponential Complexity Classes</i> , PhD Thesis Lutz, Jack H
___5242:TR:87	\$8.00	<i>Fine Grain Concurrent Computations</i> , PhD Thesis Athas, William C.
___5241:TR:87	\$3.00	<i>VLSI Mesh Routing Systems</i> , MS Thesis Flaig, Charles M
___5240:TR:87	\$2.00	<i>Submicron Systems Architecture</i> ARPA Semiannual Technical Report
___5239:TR:87	\$3.00	<i>Trace Theory and Systolic Computations</i> Rem, Martin
___5238:TR:87	\$7.00	<i>Incorporating Time in the New World of Computing System</i> , MS Thesis Poh, Hean Lee
___5236:TR:86	\$4.00	<i>Approach to Concurrent Semantics Using Complete Traces</i> , MS Thesis Van Horn, Kevin S.
___5235:TR:86	\$4.00	<i>Submicron Systems Architecture</i> ARPA Semiannual Technical Report
___5234:TR:86	\$3.00	<i>High Performance Implementation of Prolog</i> Newton, Michael O
___5233:TR:86	\$3.00	<i>Some Results on Kolmogorov-Chaitin Complexity</i> , MS Thesis Schweizer, David Lawrence
___5232:TR:86	\$4.00	<i>Cantor User Report</i> Athas, W.C. and C. L. Seitz

Caltech Computer Science Technical Reports

- ____5231:TR:86 \$2.00 *Deadlock-Free Message Routing in Multiprocessor Interconnection Networks*
Dally, William J and Charles L Seitz
current supply only: see *IEEE Transactions on Computers* vol C-36 no 5, May 1987
- ____5230:TR:86 \$24.00 *Monte Carlo Methods for 2-D Compaction*, PhD Thesis
Mosteller, R.C.
- ____5229:TR:86 \$4.00 *anaLOG - A Functional Simulator for VLSI Neural Systems*, MS Thesis
Lazzaro, John
- ____5228:TR:86 \$3.00 *On Performance of k-ary n-cube Interconnection Networks*,
Dally, Wm. J
- ____5227:TR:86 \$18.00 *Parallel Execution Model for Logic Programming*, PhD Thesis
Li, Pey-yun Peggy
- ____5221:TR:86 \$3.00 *Sync Model: A Parallel Execution Method for Logic Programming*
Li, Pey-yun Peggy and Alain J. Martin
current supply only: see *Proc SLP'86 3rd IEEE Symp on Logic Programming Sept '86*
- ____5220:TR:86 \$4.00 *Submicron Systems Architecture*
ARPA Semiannual Technical Report
- ____5215:TR:86 \$2.00 *How to Get a Large Natural Language System into a Personal Computer*,
Thompson, Bozena H. and Frederick B. Thompson
- ____5214:TR:86 \$2.00 *ASK is Transportable in Half a Dozen Ways*,
Thompson, Bozena H. and Frederick B. Thompson
- ____5212:TR:86 \$2.00 *On Seitz' Arbiter*,
Martin, Alain J
- ____5211:TR:86 \$3.00 *Self-timed FIFO: An exercise in Compiling Programs into VLSI Circuits*
Martin, Alain J
current supply only: see *HDL Description to Guaranteed Correct Circuit Design*
North-Holland, ed D. Barrione, (1986)
- ____5210:TR:86 \$2.00 *Compiling Communicating Processes into Delay-Insensitive VLSI Circuits*,
Martin, Alain
current supply only: see *Distributed Computing* v 1 no 4 (1986)
- ____5209:TR:86 \$11.00 *VLSI Architecture for Concurrent Data Structures*, PhD Thesis,
Dally, William J.
- ____5208:TR:86 \$2.00 *The Torus Routing Chip*,
Dally, William and Charles L Seitz
current supply only: see *Distr. Computing* vol 1 no 4 1986
- ____5207:TR:86 \$2.00 *Complete and Infinite Traces: A Descriptive Model of Computing Agents*,
van Horn, Kevin
- ____5205:TR:85 \$2.00 *Two Theorems on Time Bounded Kolmogorov-Chaitin Complexity*,
Schweizer, David and Yaser Abu-Mostafa
- ____5204:TR:85 \$3.00 *An Inverse Limit Construction of a Domain of Infinite Lists*,
Choo, Young-II
- ____5202:TR:85 \$15.00 *Submicron Systems Architecture*,
ARPA Semiannual Technical Report
- ____5200:TR:85 \$18.00 *ANIMAC: A Multiprocessor Architecture for Real-Time Computer Animation*, PhD thesis
Whelan, Dan
- ____5198:TR:85 \$8.00 *Neural Networks, Pattern Recognition and Fingerprint Hallucination*, PhD thesis
Mjolsness, Eric
- ____5197:TR:85 \$7.00 *Sequential Threshold Circuits*, MS thesis
Platt, John
- ____5196:TR:85 \$5.00 *ECL: An Experimental Concurrent Language*,
Athas, Bill
- ____5195:TR:85 \$3.00 *New Generalization of Dekker's Algorithm for Mutual Exclusion*,
Martin, Alain J
current supply only: see *Information Processing Letters*, **23**, 295-297 (1986)

Caltech Computer Science Technical Reports

___5194:TR:85	\$5.00	<i>Sneptree - A Versatile Interconnection Network</i> , Li, Pey-yun Peggy and Alain J Martin
___5193:TR:85	\$2.00	<i>Delay-insensitive Fair Arbiter</i> Martin, Alain J
___5190:TR:85	\$3.00	<i>Concurrency Algebra and Petri Nets</i> , Choo, Young-il
___5189:TR:85	\$10.00	<i>Hierarchical Composition of VLSI Circuits</i> , PhD Thesis Whitney, Telle
___5185:TR:85	\$11.00	<i>Combining Computation with Geometry</i> , PhD Thesis Lien, Sheue-Ling
___5184:TR:85	\$7.00	<i>Placement of Communicating Processes on Multiprocessor Networks</i> , Ms Thesis Steele, Craig
___5179:TR:85	\$3.00	<i>Sampling Deformed, Intersecting Surfaces with Quadtrees</i> , MS Thesis, Von Herzen, Brian P.
___5178:TR:85	\$9.00	<i>Submicron Systems Architecture</i> , ARPA Semiannual Technical Report
___5177:TR:85	\$4.00	<i>Hot-Clock nMOS</i> , Proc. 1985 Chapel Hill Conference on VLSI, pp 1-17 Seitz, Charles, A H Frey, S Mattisson, S D Rabin, D A Speck, and J L A van de Snepscheut
___5174:TR:85	\$7.00	<i>Balanced Cube: A Concurrent Data Structure</i> , Dally, William J and Charles L Seitz
___5172:TR:85	\$6.00	<i>Combined Logical and Functional Programming Language</i> , Newton, Michael
___5168:TR:84	\$3.00	<i>Object Oriented Architecture</i> , Dally, Bill and Jim Kajiya
___5165:TR:84	\$4.00	<i>Customizing One's Own Interface Using English as Primary Language</i> , Thompson, B H and Frederick B Thompson
___5164:TR:84	\$13.00	<i>ASK French - A French Natural Language Syntax</i> , MS Thesis Sanouillet, Remy
___5160:TR:84	\$7.00	<i>Submicron Systems Architecture</i> , ARPA Semiannual Technical Report
___5158:TR:84	\$6.00	<i>VLSI Architecture for Sound Synthesis</i> , Wawrzynek, John and Carver Mead
___5157:TR:84	\$15.00	<i>Bit-Serial Reed-Solomon Decoders in VLSI</i> , PhD Thesis Whiting, Douglas
___5148:TR:84	\$4.00	<i>Fair Mutual Exclusion with Unfair P and V Operations</i> , Martin, Alain and Jerry Burch current supply only: see <i>Information Processing Letters</i> , 21 , 97-100, (1985)
___5147:TR:84	\$4.00	<i>Networks of Machines for Distributed Recursive Computations</i> , Martin, Alain and Jan van de Snepscheut
___5143:TR:84	\$5.00	<i>General Interconnect Problem</i> , MS Thesis Ngai, John
___5140:TR:84	\$5.00	<i>Hierarchy of Graph Isomorphism Testing</i> , MS Thesis Chen, Wen-Chi
___5139:TR:84	\$4.00	<i>HEX: A Hierarchical Circuit Extractor</i> , MS Thesis Oyang, Yen-Jen
___5137:TR:84	\$7.00	<i>Dialogue Designing Dialogue System</i> , PhD Thesis Ho, Tai-Ping
___5136:TR:84	\$5.00	<i>Heterogeneous Data Base Access</i> , PhD Thesis Papachristidis, Alex
___5135:TR:84	\$7.00	<i>Toward Concurrent Arithmetic</i> , MS Thesis Chiang, Chao-Lin
___5134:TR:84	\$2.00	<i>Using Logic Programming for Compiling APL</i> , MS Thesis Derby, Howard

Caltech Computer Science Technical Reports

___5133:TR:84	\$13.00	<i>Hierarchical Timing Simulation Model for Digital Integrated Circuits and Systems</i> , PhD Thesis Lin, Tzu-mu
___5132:TR:84	\$10.00	<i>Switch Level Fault Simulation of MOS Digital Circuits</i> , MS Thesis Schuster, Mike
___5130:TR:84	\$3.00	<i>LOG The Chipmunk Logic Simulator User's Guide</i> , Gillespie, Dave
___5129:TR:84	\$5.00	<i>Design of the MOSAIC Processor</i> , MS Thesis Lutz, Chris
___5128:TM:84	\$3.00	<i>Linguistic Analysis of Natural Language Communication with Computers</i> , Thompson, Bozena H
___5125:TR:84	\$6.00	<i>Supermesh</i> , MS Thesis Su, Wen-king
___5124:TR:84	\$4.00	<i>Probe: An Addition to Communication Primitives</i> , Martin, Alain current supply only: see <i>Information Processing Letters</i> , 20, no 3, (1985)
___5123:TR:84	\$14.00	<i>Mossim Simulation Engine Architecture and Design</i> , Dally, Bill
___5122:TR:84	\$8.00	<i>Submicron Systems Architecture</i> , ARPA Semiannual Technical Report
___5120:TM:84	\$1.00	<i>Mathematical Approach to Modeling the Flow</i> , Johnsson, Lennart and Danny Cohen
___5119:TM:84	\$1.00	<i>Integrative Approach to Engineering Data and Automatic Project Coordination</i> , Segal, Richard
___5118:TR:84	\$2.00	<i>SMART User's Guide</i> , Ngai, John
___5114:TM:84	\$3.00	<i>ASK As Window to the World</i> , Thompson, Bozena, and Fred Thompson
___5113:TR:84	\$4.00	<i>WoLery</i> , Mead, Carver A
___5112:TR:83	\$22.00	<i>Parallel Machines for Computer Graphics</i> , PhD Thesis Ulner, Michael
___5106:TM:83	\$1.00	<i>Ray Tracing Parametric Patches</i> , Kajiya, James T
___5105:TR:83	\$2.00	<i>Memory Management in the Programming Language ICL</i> , Wawrzynek, John
___5104:TR:83	\$9.00	<i>Graph Model and the Embedding of MOS Circuits</i> , MS Thesis Ng, Tak-Kwong
___5103:TR:83	\$7.00	<i>Submicron Systems Architecture</i> , ARPA Semiannual Technical Report
___5102:TR:83	\$2.00	<i>Experiments with VLSI Ensemble Machines</i> , Seitz, Charles L
___5101:TM:83	\$1.00	<i>Concurrent Fault Simulation of MOS Digital Circuits</i> , Bryant, Randal E
___5099:TM:83	\$1.00	<i>VLSI and the Foundations of Computation</i> , Mead, Carver
___5098:TM:83	\$2.00	<i>New Techniques for Ray Tracing Procedurally Defined Objects</i> , Kajiya, James T
___5097:TR:83	\$4.00	<i>Design of a Self-timed Circuit for Distributed Mutual Exclusion</i> , Martin, Alain J current supply only: see <i>Proc. Chapel Hill Conf. on VLSI</i> , 245-259, May 1985
___5094:TR:83	\$2.00	<i>Stochastic Estimation of Channel Routing Track Demand</i> , Ngai, John

Caltech Computer Science Technical Reports

___5093:TR:83	\$1.00	<i>Design of the MOSAIC Element,</i> Lutz, Chris, Steve Rabin, Chuck Seitz and Don Speck
___5092:TM:83	\$2.00	<i>Residue Arithmetic and VLSI,</i> Chiang, Chao-Lin and Lennart Johnsson
___5091:TR:83	\$2.00	<i>Race Detection in MOS Circuits by Ternary Simulation,</i> Bryant, Randal E
___5090:TR:83	\$9.00	<i>Space-Time Algorithms: Semantics and Methodology,</i> PhD Thesis Chen, Marina Chien-mei
___5089:TR:83	\$10.00	<i>Signal Delay in General RC Networks with Application to Timing Simulation of Digital Integrated Circuits,</i> Lin, Tzu-Mu and Carver A Mead
___5086:TR:83	\$4.00	<i>VLSI Combinator Reduction Engine,</i> MS Thesis Athas, William C Jr
___5084:TM:83	\$3.00	<i>Tree Machine: An Evaluation of Strategies for Reducing Program Loading Time,</i> Li, Pey-yun Peggy, and Lennart Johnsson
___5082:TR:83	\$10.00	<i>Hardware Support for Advanced Data Management Systems,</i> PhD Thesis Neches, Philip
___5081:TR:83	\$4.00	<i>RTsim - A Register Transfer Simulator,</i> MS Thesis Lam, Jimmy
___5080:TR:83	\$4.00	<i>Distributed Mutual Exclusion on a Ring of Processes,</i> Martin, Alain current supply only: see <i>Science of Computer Programming</i> , 5, (1985)
___5079:TR:83	\$2.00	<i>Highly Concurrent Algorithms for Solving Linear Systems of Equations,</i> Johnsson, Lennart
___5078:TR:83	\$5.00	<i>Submicron Systems Architecture,</i> ARPA Semiannual Technical Report
___5075:TR:83	\$2.00	<i>General Proof Rule for Procedures in Predicate Transformer Semantics,</i> Martin, Alain current supply only: see <i>Acta Informatica</i> 20, 301-313, (1983)
___5074:TR:83	\$10.00	<i>Robust Sentence Analysis and Habitability,</i> Trawick, David
___5073:TR:83	\$12.00	<i>Automated Performance Optimization of Custom Integrated Circuits,</i> PhD Thesis Trimberger, Steve
___5068:TM:83	\$1.00	<i>Hierarchical Simulator Based on Formal Semantics,</i> Proc Third Caltech Conf on VLSI Chen, Marina and Carver Mead
___5065:TR:82	\$3.00	<i>Switch Level Model and Simulator for MOS Digital Systems,</i> Bryant, Randal E
___5055:TR:82	\$5.00	<i>FIFO Buffering Transceiver: A Communication Chip Set for Multiprocessor Systems,</i> MS Thesis Ng, Charles H
___5054:TM:82	\$3.00	<i>Introducing ASK, A Simple Knowledgeable System,</i> Conf on App'l Natural Language Processing Thompson, Bozena H and Frederick B Thompson
___5052:TR:82	\$8.00	<i>Submicron Systems Architecture,</i> ARPA Semiannual Technical Report
___5051:TM:82	\$2.00	<i>Knowledgeable Contexts for User Interaction,</i> Proc Nat'l Computer Conference Thompson, Bozena, Frederick B Thompson, and Tai-Ping Ho
___5047:TR:82	\$3.00	<i>Torus: An Exercise in Constructing a Processing Surface,</i> Proc 2nd Caltech Conference on VLSI Martin, Alain
___5046:TR:82	\$3.00	<i>in Axiomatic Definition of Synchronization Primitives</i> Martin, Alain current supply only: see <i>Acta Informatica</i> 16, pp 219-235 (1981)
___5045:TM:82	\$3.00	<i>in Distributed Implementation Method for Parallel Programming</i> Martin, Alain see <i>Proc Information Processing '80</i>

Caltech Computer Science Technical Reports

___5044:TR:82	\$10.00	<i>Hierarchical Nets: A Structured Petri Net Approach to Concurrency,</i> Choo, Young-Il
___5038:TM:82	\$4.00	<i>New Channel Routing Algorithm,</i> Chan, Wan S
___5035:TR:82	\$9.00	<i>Type Inference in a Declarationless, Object-Oriented Language,</i> MS Thesis Holstege, Eric
___5034:TR:82	\$12.00	<i>Hybrid Processing,</i> PhD Thesis Carroll, Chris
___5033:TR:82	\$4.00	<i>MOSSIM II: A Switch-Level Simulator for MOS LSI User's Manual,</i> Schuster, Mike, Randal Bryant and Doug Whiting
___5029:TM:82	\$4.00	<i>POOH User's Manual,</i> Whitney, Telle
___5021:TR:82	\$5.00	<i>Earl: An Integrated Circuit Design Language,</i> MS Thesis Kingsley, Chris
___5018:TM:82	\$2.00	<i>Filtering High Quality Text for Display on Raster Scan Devices,</i> Kajiya, Jim and Mike Ullner
___5017:TM:82	\$2.00	<i>Ray Tracing Parametric Patches,</i> Kajiya, Jim
___5016:TR:82	\$4.00	<i>Bristle Blocks - Scrutinized and Analyzed,</i> McNair, Richard and Monroe Miller
___5015:TR:82	\$15.00	<i>VLSI Computational Structures Applied to Fingerprint Image Analysis,</i> Megdal, Barry
___5014:TR:82	\$15.00	<i>Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture,</i> PhD Thesis Lang, Charles R Jr
___5012:TM:82	\$2.00	<i>Switch-Level Modeling of MOS Digital Circuits,</i> Bryant, Randal
___5001:TR:82	\$2.00	<i>Minimum Propagation Delays in VLSI ,</i> IEEE J Solid State Circuits Mead, Carver, and Martin Rem
___5000:TR:82	\$6.00	<i>Self-Timed Chip Set for Multiprocessor Communication,</i> MS Thesis Whiting, Douglas
___4777:TR:82	\$7.00	<i>Techniques for Testing Integrated Circuits,</i> PhD Thesis DeBenedictis, Erik P
___4724:TR:82	\$2.00	<i>Concurrent, Asynchronous Garbage Collection Among Cooperating Processors,</i> Lang, Charles R
___4716:TM:82	\$4.00	<i>Rectangular Area Filling Display System Architecture,</i> Whelan, Dan
___4684:TR:82	\$3.00	<i>Characterization of Deadlock Free Resource Contentions,</i> Chen, Marina, Martin Rem, and Ronald Graham
___4675:TR:81	\$7.00	<i>Switching Dynamics,</i> MS Thesis Lewis, Robert K
___4655:TR:81	\$20.00	<i>Proc Second Caltech Conf on VLSI,</i> Seitz, Charles, ed.
___4654:TR:81	\$12.00	<i>Versatile Ethernet Interface,</i> MS Thesis Whelan, Dan
___4653:TR:81	\$10.00	<i>Toward A Theorem Proving Architecture,</i> MS Thesis Lien, Sheue-Ling
___4618:TM:81	\$5.00	<i>Tree Machine Operating System,</i> Li, Peggy
___4600:TM:81	\$3.00	<i>Notation for Designing Restoring Logic Circuitry,</i> Proc Second Caltech Conf on VLSI Rem, Martin, and Carver Mead
___4530:TR:81	\$20.00	<i>Silicon Compilation,</i> PhD Thesis Johannsen, Dave

Caltech Computer Science Technical Reports

4527:TR:81	\$11.00	<i>Communicative Databases</i> , PhD Thesis Yu, Kwang-I
4521:TR:81	\$8.00	<i>Lambda Logic</i> , MS Thesis Rudin, Leonid
4517:TR:81	\$7.00	<i>Serial Log Machine</i> , MS Thesis Li, Peggy
4407:TM:82	\$3.00	<i>Experimental Composition Tool</i> , Mosteller, Richard C
4332:TR:81	\$3.00	<i>RLAP, Version 1.0, A Chip Assembly Tool</i> , Mosteller, R
4320:TR:81	\$7.00	<i>Hierarchical Design Rule Checker</i> , MS Thesis Whitney, Telle
4317:TR:81	\$10.00	<i>REST - A Leaf Cell Design System</i> , MS Thesis Mosteller, Richard C
4298:TR:81	\$7.00	<i>From Geometry to Logic</i> , MS Thesis Lin, Tzu-mu
4287:TR:81	\$3.00	<i>Computational Arrays for Band Matrix Equations</i> Johnsson, Lennart
4204:TR:78	\$8.00	<i>16-Bit LSI Digital Multiplier</i> , EE Thesis Masumoto, R T
4191:TR:81	\$4.00	<i>Towards A Formal Treatment of VLSI Arrays</i> , Proc Second Caltech Conf on VLSI Johnsson, Lennart S, Uri Weiser, D Cohen, and Alan L Davis
4128:TM:81	\$2.00	<i>Shifting to a Higher Gear in a Natural Language System</i> , Thompson, Fred and B Thompson
3975:TM:80	\$3.00	<i>Rapidly Extendable Natural Language</i> , Thompson, B H and Fred B Thompson
3762:TR:80	\$8.00	<i>Software Design System</i> , PhD Thesis Hess, Gideon
3761:TR:80	\$7.00	<i>Fault Tolerant Integrated Circuit Memory</i> , PhD Thesis Barton, Tony
3760:TR:80	\$10.00	<i>Tree Machine: A Highly Concurrent Computing Environment</i> , PhD Thesis Browning, Sally
3759:TR:80	\$10.00	<i>Homogeneous Machine</i> , PhD Thesis Locanthi, Bart
3710:TR:80	\$10.00	<i>Understanding Hierarchical Design</i> , PhD Thesis Rowson, James
3364:TR:79	\$8.00	<i>Stack Data Engine</i> , Efland, G and R C Mosteller
3340:TR:79	\$26.00	<i>Proc. Caltech Conference on VLSI (1979)</i> , Seitz, Charles, ed
2276:TM:78	\$12.00	<i>Language Processor and a Sample Language</i> , Ayres, Ron
2275:TR:70	\$17.00	<i>Formal Methods in the Foundations of Science</i> , PhD Thesis Randall, David Lawrence

Caltech Computer Science Technical Reports

Please fill in your name, address and amount enclosed below:

name _____

Address _____

City _____ State _____ Zip _____ Country _____

Amount enclosed \$ _____

_____ Please check here if you wish to be included on our mailing list

_____ Please check here for any change of address

Return this form to: Computer Science Library, 256-80, Caltech, Pasadena CA 91125

A Synthesis Method for Self-Timed VLSI Circuits

Alain J. Martin

**The research described in this paper was sponsored by
the Defense Advanced Research Projects Agency, ARPA Order No. 6202,
and monitored by the office of Naval Research
under contract number N00014-87-K-0745**

©California Institute of Technology, 1987

**Department of Computer Science
California Institute of Technology
Pasadena, CA 91125**

5256:TR:87

**Published in: *Proc. 1987 IEEE International Conference on
Design: VLSI in Computers & Processors*
ICCD '87, Rye Town Hilton, Rye Brook, New York
October 5 - October 8, 1987; pp 224-229
IEEE Computer Society Press**

A Synthesis Method for Self-timed VLSI Circuits

Alain J. Martin

Department of Computer Science
California Institute of Technology
Pasadena CA 91125, USA

1. Introduction

With chip size reaching 1 million transistors, the need for high-level design of circuits becomes compelling. The main stumbling block in the development of design methods for VLSI algorithms is to find an interface that provides a good separation of the physical and algorithmic concerns. Among the physical issues, timing is the most critical, since it is not only essential to the real-time behavior of a circuit, but also to its logical correctness if synchronous techniques are used.

Synchronous techniques are detrimental to the use of high-level design methods because they don't "scale well": a circuit may cease to function correctly when its feature sizes are scaled down to smaller dimensions. Further, with the increasing size of circuits, it becomes more and more difficult to distribute safely a clock signal across a chip, and the restrictions attached to wire lengths in order to maintain certain timing properties add extra complication to the already difficult layout problem.

For all those reasons, self-timed techniques (as defined in [10]) are particularly attractive for high-level VLSI design [9]. We propose a synthesis method for self-timed circuits in which the computation is initially described as a set of communicating processes in the notation of [3], which is similar to C.A.R. Hoare's CSP [2] but augmented with the *probe* construct. This first description is the reference solution, which has to be proved correct. The program is then compiled into a self-timed circuit by applying a series of semantics-preserving transformations. Hence the circuit obtained is correct by construction.

Unlike most silicon compilation methods and hardware description languages, the method leads to efficient circuits. It has been applied with "hand compilation" to a series of difficult self-timed design problems, such as distributed mutual exclusion, fair arbitration, routing automata, with great success. Actually, the method, applied by a person in a mechanical way, will typically produce better results than the most experienced designers can produce. The main reason for the efficiency of the method is that, rather than going in one step from the program notation to the circuit, the designer applies a series of transformations to the original program. At each level of the transformation, powerful algebraic manipulations can be performed leading to important optimizations in terms of speed or area.

We shall first present the program notation and the VLSI operators that constitute the "object code". We then describe the four steps of the compilation and illustrate the method with one sizeable example, the construction of a stack. We shall conclude that this technique can be used for high quality and high complexity designs, fully automated from a provably correct high-level description. (For a more complete description of the method, see [4], [5], [6], and [7].)

2. The program notation

The language used for the high-level description is close to C.A.R. Hoare's CSP [2]. We give only a very informal definition of the constructs used in this paper.

- i) $b \uparrow$ stands for $b := \text{true}$, $b \downarrow$ stands for $b := \text{false}$.
- ii) The execution of the *selection* command (generalized IF-statement) $[G_1 \rightarrow S_1 \mid \dots \mid G_n \rightarrow S_n]$, where G_1 through G_n are Boolean expressions, and S_1 through S_n are program parts, (G_i is called a "guard", and $G_i \rightarrow S_i$ a "guarded command") amounts to the execution of an arbitrary S_i for which G_i holds. If $\neg(G_1 \vee \dots \vee G_n)$ holds, the execution of the command is suspended until $(G_1 \vee \dots \vee G_n)$ holds.
- iii) For atomic actions x and y , " x, y " stands for the execution of x and y in any order.
- iv) $[G]$ where G is a Boolean, stands for $[G \rightarrow \text{skip}]$, and thus for "wait until G holds". (Hence, " $[G]; S$ " and $[G \rightarrow S]$ are equivalent.)
- v) $*[S]$ stands for "repeat S forever".
- vi) From ii) and iii), the operational description of the statement $*[[G_1 \rightarrow S_1 \mid \dots \mid G_n \rightarrow S_n]]$ is "repeat forever: wait until some G_i holds; execute an S_i for which G_i holds".

Communicating processes

A concurrent computation is described as a set of processes communicating with each other by communication actions on channels (no shared variables). When no messages are transmitted, communication on a channel is reduced to synchronisation signals. The name of the channel is then sufficient for identifying a communication action.

If two processes p_1 and p_2 share a channel named X in p_1 and Y in p_2 , at any time the completion of the n th X -action "coincides" with the completion of the n th Y -action. If, for example, p_1 reaches the n th X -action before p_2 reaches the n th Y -action, the completion of X is suspended until p_2 reaches Y . The X -action is then said to be *pending*.

Probe

Instead of the usual selection mechanism by which a set of pending communication actions can be selected for execution, we provide a general Boolean command on channels, called the *probe*. In process p_1 , the probe command \bar{X} has the same value as the predicate "A communication action Y is pending in p_2 ".

Hence the guarded command $\bar{X} \rightarrow X$ guarantees that the X -action is not suspended. And a construct of the form $[\bar{X} \rightarrow X \mid \bar{Y} \rightarrow Y]$ can be used for selection.

3. The "object code"

In standard digital VLSI design, the MOS transistor is idealized as an on/off switch. Unfortunately, the switch model is too crude, ignoring too many electrical phenomena that play

an important role in the functioning of the circuit. Therefore, trying to carry the discrete model of a computation down to the transistor level is very likely to lead either to incorrect implementations or to a too complicated model of the computation. A crucial decision in the development of our method has been to choose an "object code" at a higher level than the transistor. We have chosen to construct a notation that provides the weakest possible form of control structure and smallest number of program constructs. In fact, the notation contains exactly one construct, the *production rule*, and is therefore called the "production-rule set notation".

This minimal notation has been chosen so that i) it has sound semantics, ii) any non-terminating program can be compiled into production rules, iii) the transformation into a circuit is straightforward.

In fact, we consider the production-rule set as the canonical representation of a circuit. This representation can be decomposed into several equivalent networks of gates depending on the set of building blocks used, but the production-rule set represents the circuit independently of the gate implementations.

4. Production rules

Production rules can be seen as a weaker form of guarded commands. Consider the production rule $G \rightarrow S$

- S is either a simple assignment, or an unordered list " s_1, s_2, s_3, \dots " of simple assignments, where a simple assignment is the assignment of true or false to a single Boolean variable.
- G is a Boolean expression, called the guard of the production rule. If G holds, the correct execution of S is guaranteed only if G remains invariantly true until the completion of S . We say that G must be *stable*.

A *production rule set* is an unordered set (a collection) of production rules. Consider the canonical production rule set PRS :

$$\begin{aligned} G_1 &\rightarrow S_1 \\ G_2 &\rightarrow S_2 \\ &\dots \\ G_n &\rightarrow S_n \end{aligned}$$

- Unlike the guarded commands of a selection or a repetition, the mutual exclusion among the different production rules of a set is not part of the semantics of the construct. The correct execution of a production rule set is guaranteed only if *interfering* production rules are mutually exclusive. Two production rules are said to be interfering when their right-hand sides share a variable. Each process will be implemented as a p.r.s. such that exactly one p.r. is fireable at any time, hence enforcing non-interference.

- If stability of the guards and mutual exclusion among interfering production rules are guaranteed, the production rule set PRS is semantically equivalent to the non-terminating repetition $*[[GCS]]$, where GCS is the guarded command set syntactically identical to PRS . Stability of the guards is essential to guarantee the absence of races and hazards. When stability cannot be enforced, a special operator called "synchronizer" has to be used. When mutual exclusion cannot be enforced, a special operator called "arbiter" has to be used. These two operators are not needed in this paper.

We implement a p.r.s. by decomposing it into a collection of production rule sets each of which has a known VLSI implementation. Those primitive production rule sets correspond to logic gates or standard VLSI cells that are our ultimate building blocks.

The set of operators with which we want to build our circuits is not unique. The descriptions of the operators used in this paper in terms of their production rules and their logic symbols are as follows.

The "and":

$$(x, y) \Delta z \equiv \begin{aligned} &x \wedge y \rightarrow z \uparrow \\ &\neg x \vee \neg y \rightarrow z \downarrow \end{aligned}$$

The "or":

$$(x, y) \vee z \equiv \begin{aligned} &x \vee y \rightarrow z \uparrow \\ &\neg x \wedge \neg y \rightarrow z \downarrow \end{aligned}$$

The wire:

$$x \underline{} y \equiv \begin{aligned} &x \rightarrow y \uparrow \\ &\neg x \rightarrow y \downarrow \end{aligned}$$

The fork:

$$x \underline{} (y, z) \equiv \begin{aligned} &x \rightarrow y \uparrow, z \uparrow \\ &\neg x \rightarrow y \downarrow, z \downarrow \end{aligned}$$

The C-element:

$$(x, y) \underline{} z \equiv \begin{aligned} &x \wedge y \rightarrow z \uparrow \\ &\neg x \wedge \neg y \rightarrow z \downarrow \end{aligned}$$

The asymmetric C-element:

$$(x; y) \underline{} z \equiv \begin{aligned} &x \wedge y \rightarrow z \uparrow \\ &\neg x \rightarrow z \downarrow \end{aligned}$$

The "flip-flop":

$$(x; y) \underline{} z \equiv \begin{aligned} &x \rightarrow z \uparrow \\ &y \rightarrow z \downarrow \end{aligned}$$

A negated input or output is represented on the figures by a small circle on the corresponding port. A wire with its input negated is an inverter. A cell with a negated input is considered as one cell, and not as the composition of an inverter and a cell.

5. The compilation method

Process decomposition

The first step of the compilation, called "process decomposition", consists in replacing a process by several semantically equivalent processes. The purpose of the decomposition is to obtain a process representation of the program in which the right-hand side of each guarded command is a straight-line program, i.e., consists only of simple assignments and communication commands, composed by semi-colons and commas. Process decomposition is applied repeatedly until the right-hand side of each guarded command is a straight-line program. Process decomposition plays an important role in the compilation of large programs. We won't need it in the example treated here. See [5] for a typical use of this transformation.

Handshaking expansion

The implementation of communication, called "handshaking expansion", replaces each channel by a pair of wire-operators and each communication action by its implementation in terms of a "four-phase handshaking" protocol. Channel (X, Y) is implemented by the two wires $(x_0 \underline{} y_1)$ and $(y_0 \underline{} x_1)$.

Initially, x_0 , x_1 , y_0 , and y_1 are false. For a matching pair (X, Y) of actions, the implementation is not symmetrical in X and Y . One action is called *active* and the other one *passive*. The four-phase implementation with X active and Y passive is:

$$X \equiv x_0 \uparrow; [x_1]; x_0 \downarrow; [\neg x_1] \quad (1)$$

$$Y \equiv [yi]; yo \uparrow; [\neg yi]; yo \downarrow \quad (2)$$

When no action of a matching pair is probed, the choice of which one should be active and which one passive is arbitrary, but a choice has to be made. The choice can be important for the composition of identical circuits. A simple rule is that for a given channel (X, Y) , all actions at one side are active and all actions at the other side are passive. If \bar{X} is used, all X -actions are passive—with the obvious restriction that \bar{Y} cannot be used in the same program. The implementation of the probe is simply:

$$\begin{aligned} \bar{X} &\equiv xi \\ \bar{Y} &\equiv yi \end{aligned} \quad (3)$$

A probed communication action $\bar{X} \rightarrow \dots X$ is implemented:

$$xi \rightarrow \dots xo \uparrow; [\neg xi]; xo \downarrow.$$

Reshuffling

Consider the handshaking expansion of program p according to (1), (2), and (3). Provided that the cyclic order of the four handshaking actions of a communication command is respected, the last two actions of this command can be inserted at any place in p without invalidating the semantics of the communication involved. However, modifying the order of these two actions relatively to other actions of p may introduce deadlock. The possibility to reshuffle the second half of the handshaking sequence, plays an important role in the compilation method as a source of algebraic manipulations.

Production rule expansion

The next step is to compile the handshaking expansion of the program into a set of production rules from which all explicit sequencing has been removed. This is the most difficult step in particular because it requires, in all but trivial cases, the introduction of state variables to identify each state of the computation uniquely.

Operator reduction

The last step, called "operator reduction", consists in identifying sets of production rules in the program with sets of production rules describing operators. The non-trivial part in this step is called "symmetrization". It is used for transforming the guards of the production rules so as to make them 'look like' the guards of operators. After this last step, the program has been replaced by a network of operators for which standard cells exist. (We have constructed a cell library of self-timed elements in SC MOS technology. Since many cells are parametrized, the library is extendable.)

6. Example: single variable register

Consider the following process that provides read and write access to a simple boolean variable x :

$$*[[\bar{P} \rightarrow P?x \mid \bar{Q} \rightarrow Q!x]] \quad (4)$$

where $\neg\bar{P} \vee \neg\bar{Q}$ holds at any time, i.e., read and write requests exclude each other in time.

Handshaking expansion

The handshaking expansion of (4) uses the "double-rail" technique: the Boolean value of x is encoded on two wires, one

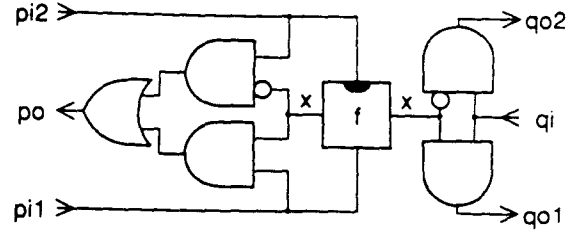


Figure 1: Single-bit register

for the value true and one for the value false. Each guarded command of (1) is expanded to two guarded commands:

$$\begin{aligned} &*[[pi_1 \rightarrow x \uparrow; [x]; po \uparrow; [\neg pi_1]; po \downarrow \\ &\quad | pi_2 \rightarrow x \downarrow; [\neg x]; po \uparrow; [\neg pi_2]; po \downarrow \\ &\quad | x \wedge qi \rightarrow qo_1 \uparrow; [\neg qi]; qo_1 \downarrow \\ &\quad | \neg x \wedge qi \rightarrow qo_2 \uparrow; [\neg qi]; qo_2 \downarrow \\ &\quad]]. \end{aligned} \quad (5)$$

Production rule expansion

The production-rule expansion of the first two guarded commands gives:

$$\begin{aligned} pi_1 &\mapsto x \uparrow \\ pi_1 \wedge x &\mapsto po \uparrow \\ \neg pi_1 &\mapsto po \downarrow \\ pi_2 &\mapsto x \downarrow \\ pi_2 \wedge \neg x &\mapsto po \uparrow \\ \neg pi_2 &\mapsto po \downarrow. \end{aligned}$$

The first and fourth p.r.'s correspond to the flip-flop: $(pi_1; pi_2) \not\equiv x$. The other p.r.'s can be transformed into:

$$\begin{aligned} (pi_1 \wedge x) \vee (pi_2 \wedge \neg x) &\mapsto po \uparrow \\ (\neg pi_1 \vee \neg x) \vee (\neg pi_2 \vee x) &\mapsto po \downarrow \end{aligned}$$

which is the definition of the IF-cell $(pi_1; pi_2; x) \triangle po$. This set of p.r.'s can also be implemented as:

$$\begin{aligned} (pi_1, x) &\triangle po_1 \\ (pi_2, \neg x) &\triangle po_2 \\ (po_1, po_2) &\triangle po. \end{aligned}$$

The production-rule expansion of the last two guarded commands of (5) gives:

$$\begin{aligned} x \wedge qi &\mapsto qo_1 \uparrow \\ \neg x \vee \neg qi &\mapsto qo_1 \downarrow \\ \neg x \wedge qi &\mapsto qo_2 \uparrow \\ x \vee \neg qi &\mapsto qo_2 \downarrow, \end{aligned}$$

which corresponds to the two operators $(x, qi) \triangle qo_1$ and $(\neg x, qi) \triangle qo_2$. The circuit is represented in Figure 1.

7. The lazy stack

A lazy stack is one in which the full elements, i.e., the elements of the stack that contain a piece of data, are not necessarily contiguous. For instance, after a "pop" operation removes a data portion from the top element of the stack, the hole created in the top element is not filled even if some other element of the stack contains a data portion. Obviously, we must record

whether a stack element is full or empty. In the implementation given in [3], a Boolean variable is used for this purpose. Here we shall use a different coding: a stack element is described as two programs—one for the empty case, one for the full case—which call each other in a mutually recursive way.

We restrict ourselves to Boolean data portions. A data portion is added to a stack element by a command on the input channel "in". A data portion is removed from a stack element by a command on the output channel "out". We assume that the environment never attempts to add portions to a full stack nor to remove portions from an empty stack. Hence a request to remove a portion from an empty stack causes the element to obtain the next data portion from the "rest of the stack". Such an action uses the input channel "get". Similarly, a request to add a portion to a full element causes the element to push the portion it contains to the "rest of the stack". Such an action uses the output channel "put".

The program for the empty stack element is called E . The program for the full stack element is called F . We have

$$\begin{aligned} E &\equiv [\overline{in} \rightarrow in?x; F \\ &\quad \overline{out} \rightarrow get?x; out!x; E \\ &\quad] \\ F &\equiv [\overline{in} \rightarrow put!x; in?x; F \\ &\quad \overline{out} \rightarrow out!x; E \\ &\quad] \end{aligned} \quad (6)$$

The initialization of an empty stack element is a call of E . The initialization of a full stack element is a call of F .

8. Implementation of the control part

Let us first implement the "control part" of the program, i.e., the programs E and F from which message communication has been removed. We assume that the stack is empty initially. Instead of using mutual recursion, we use (what may look like) a slightly less symmetrical coding of (6): we introduce the channel (t, t') and call F from within E by the usual construction of process decomposition. We get

$$\begin{aligned} E &\equiv *[[\overline{in} \rightarrow in; t \\ &\quad \overline{out} \rightarrow get; out \\ &\quad]] \\ F &\equiv *[[t' \wedge \overline{in} \rightarrow put; in \\ &\quad t' \wedge \overline{out} \rightarrow out; t' \\ &\quad]] \end{aligned} \quad (7)$$

In the handshaking expansion, the choice of active and passive communications is entirely dictated by the occurrence of the probes. We get

$$\begin{aligned} E &\equiv \\ &*[[\neg ti \wedge ini \rightarrow ino \uparrow; [\neg ini]; ino \downarrow; to \uparrow; [ti]; to \downarrow \\ &\quad \neg ti \wedge outi \rightarrow geto \uparrow; [geti]; geto \downarrow; [\neg geti]; outo \uparrow; [\neg outi]; outo \downarrow \\ &\quad]] \\ F &\equiv \\ &*[[ti' \wedge ini \rightarrow puto \uparrow; [puti]; puto \downarrow; [\neg puti]; ino \uparrow; [\neg ini]; ino \downarrow \\ &\quad ti' \wedge outi \rightarrow outo \uparrow; [\neg outi]; outo \downarrow; to' \uparrow; [\neg ti']; to' \downarrow \\ &\quad]] \end{aligned}$$

9. Compilation of E

The first guarded command of E is a standard passive-active buffer element implemented as an active-active buffer composed with a passive-passive adaptor (Fig. 2.a). The second guarded

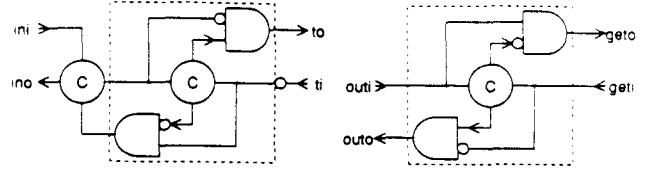


Figure 2: The two guarded commands of E

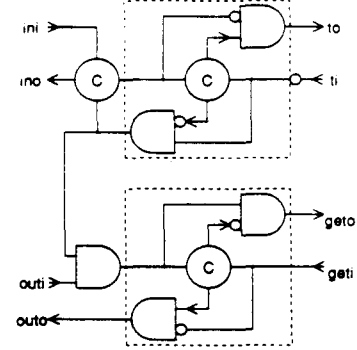


Figure 3: Implementation of E

command is a standard stack element implemented as an active-active buffer with input $outi$ inverted (Fig. 2.b). The active-active buffer is a standard cell called a D -element.

Next, we have to enforce mutual exclusion between the two guarded commands of E . Since in and out are mutually exclusive, it suffices to guarantee that when in is completed in the first guarded command, the second guarded command cannot start until t is completed. In order to strengthen the guard of the second command with the appropriate expression, we introduce in the handshaking expansion of the first guarded command the variable z . We get

$$z \wedge ini \rightarrow ino \uparrow; z \downarrow; [\neg ino]; ino \downarrow; to \uparrow; [ti]; to \downarrow; [\neg ti]; z \uparrow$$

as the handshaking expansion of the first guarded command. Obviously, it suffices to strengthen the guard of the second guarded command with z to guarantee mutual exclusion between the two g.c.'s. We get

$$outi \wedge z \rightarrow geto \uparrow; [geti]; geto \downarrow; [\neg geti]; outo \uparrow; [\neg outi]; outo \downarrow$$

Since we can weaken $\neg outi$ as $\neg outi \vee \neg z$, the only transformation is the replacement of $outi$ by $z \wedge outi$. This gives the circuit of Figure 3 as an implementation of E .

10. Compilation of F

The compilation of the first guarded command of F is identical to that of the second command of E , with the appropriate change of variables. The compilation of the second command, however, can be drastically simplified by reshuffling. Since channel (t, t') is an internal channel, we can reshuffle the handshaking sequence of t' without deadlock. The handshaking expansion of the second guarded command becomes:

$$ti' \wedge outi \rightarrow outo \uparrow; to' \uparrow; [\neg ti' \wedge \neg outi]; outo \downarrow; to' \downarrow$$

This sequence compiles immediately into the C -element: $(ti', outi) \subseteq (outo, to')$.

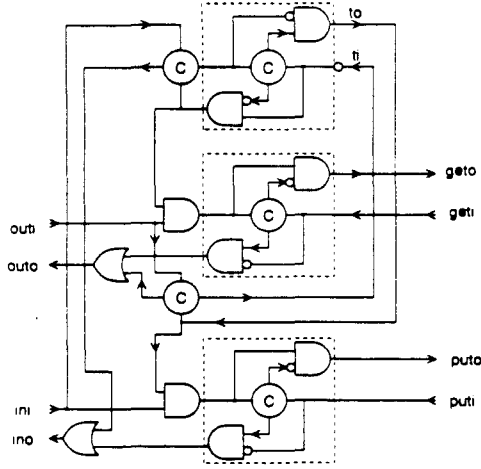


Figure 4: The control part of stack element

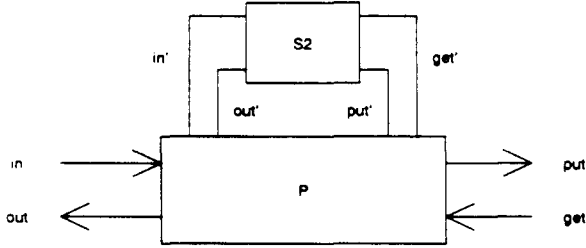


Figure 5: Adding the data path

The channels *in* and *out* are used both in *E* and *F*, so we need to merge the local copies of *in* and the local copies of *out* in the standard way. The resulting circuit for the control part of the stack element is shown in Figure 4.

11. Implementation of the data path

Let *S1* and *S2* denote program (6) and program (7), respectively. We now have to extend the implementation of *S2* so as to obtain an implementation of *S1*. We want to leave *S2* unchanged and introduce an extra "data path" process *P* such that the parallel composition of *S2* and *P* implements *S1*. More precisely, the channels *in*, *out*, *get*, *put* of *S2* are renamed *in'*, *out'*, *get'*, *put'*. *P* communicates with *S2* via the renamed channels and with the environment via *in*, *out*, *get*, *put*. (See Figure 5).

By comparing *S1* and *S2*, we derive that *P* has to implement the operations:

$$\begin{aligned} in' &\bullet in?x \\ out' &\bullet out!x \\ get' &\bullet get?x \\ put' &\bullet put!x \end{aligned}$$

where $A \bullet B$ denotes the simultaneous execution of *A* and *B*. (We can define the completion of an action so that the simultaneous execution of two actions is well-defined. The implementation of $A \bullet B$ amounts to interleaving the handshaking sequences of *A* and *B*.)

The implementation of the four actions of *P* is based on the register program constructed in Section 6. For the sake of

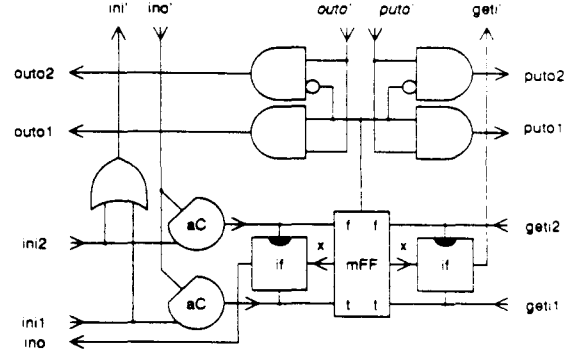


Figure 6: The data path

brevity, we omit the rest of the derivation which can be found in [8]. The entire data path is described in Figure 6.

The dual-port flip-flop used in the data path is defined as:

$$(s1, s2; t1, t2) \text{dff } x \equiv s1 \vee s2 \mapsto x \uparrow \\ t1 \vee t2 \mapsto x \downarrow$$

(By definition, at most one input is true at any time.)

12. The complete circuit

Two important optimizations are added to the design. The first one concerns the implementation of the second guard of *E*:

$$\overline{out} \rightarrow get?x; out!x.$$

We observe that, in this case, unlike all other guarded commands of (6), the value of *x* involved in the second action (*out!x*) is the same as the value of *x* involved in the first action (*get?x*). We can therefore encode the value of *x* in the handshaking expansion of the guarded command without having to use the register. The reshuffled handshaking expansion including the double-rail encoding of *x* gives:

$$\begin{aligned} \neg ti \wedge outi &\rightarrow geto \uparrow; [geti1 \rightarrow outo1 \uparrow | geti2 \rightarrow outo2 \uparrow]; \\ [\neg outi]; &geto \downarrow; [\neg geti1 \rightarrow outo1 \downarrow | \neg geti2 \rightarrow outo2 \downarrow] \end{aligned}$$

The circuit is

$$\begin{aligned} (\neg ti, outi) \triangle geto \\ geti1 \underline{w} outo1 \\ geti2 \underline{w} outo2 \end{aligned}$$

The second optimization concerns the implementation of $in' \bullet in?x$, which is more complex than that of $get' \bullet get?x$ because *in* is passive while *get* is active. We replace $in?x$ and $put!x$ by $ins; in?x$ and $outs; out!x$, respectively, with *ins* passive and *in* active, and *outs* active and *out* passive. For the output action *out*, the implementation is the same whether the channel is active or passive. The complete circuit is shown in Figure 7 with the data path extended to four bits.

13. Concluding remarks

By combining control and data, the design of a lazy stack encompasses most self-timed design issues (except for arbitration which is treated in [4] and [5]).

Let us summarize the main advantages of the method. First, the source language, in particular the use of the probe,

produces compact and efficient algorithms, which can be further "tuned" through process decomposition. Second, the handshaking expansion combined with reshuffling offers powerful algebraic manipulations. Third, the production rule notation provides a canonical representation of the circuit which is straightforward to translate in whatever set of VLSI gates is available or convenient to use. Finally, the notion of stability of a guard captures exactly the necessary and sufficient condition to avoid races and hazards.

We already have a compiler that produces about the same design fully automatically [1]. Figure 8 shows a typical layout produced by the assembler from the operator set. Each operator has a standard cell representation. The cells of a process are stacked to form a tower in which power, reset, and ground run vertically.

ACKNOWLEDGEMENTS are due to Steve Burns for his contribution to the design of the stack, and to Cal Jackson for his help in the preparation of the manuscript.

REFERENCES

- [1] Burns, S., "Automated Compilation of Concurrent Programs into Self-timed Circuits". Caltech Computer Science Master's Thesis, in preparation, (1987).
- [2] Hoare, C.A.R., "Communicating Sequential Processes". *Comm. ACM* 21, 8, pp. 666-677 (August 1978).

- [3] Martin, A.J., "The Probe: an Addition to Communication Primitives", *Information Processing Letters* 20, pp. 125-130 (1985).
- [4] Martin, A.J., "The Design of a Self-Timed Circuit for Distributed Mutual Exclusion", *Proc. 1985 Chapel Hill Conference on VLSI*, ed. Henry Fuchs, pp. 247-260 (1985).
- [5] Martin, A.J., "A Delay-Insensitive Fair Arbiter", Caltech Computer Science Technical Report 5193:TR:85 (1985).
- [6] Martin, A.J., "Compiling Communicating Processes into Delay Insensitive VLSI circuits", in *Distributed Computing*, vol. 1, no 3, 1986.
- [7] Martin, A.J., "Self-timed FIFO: an Exercise in Compiling Programs into Circuits" in *From HDL description to guaranteed correct circuit design*, North-Holland, ed. Dominique Borrione (1986).
- [8] Martin, A.J., "Implementation of Communication with Message-passing", Caltech Computer Science Technical Report 5245:TR:87 (1987).
- [9] Mead, C. and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading MA (1980).
- [10] Seitz, C.L., "System Timing", Chapter 7 in Mead & Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading MA (1980).

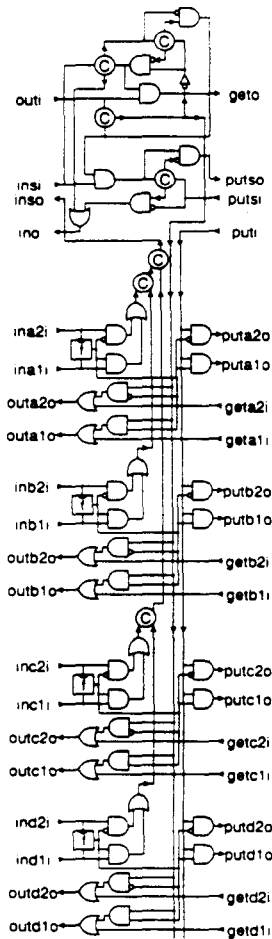


Figure 7: Stack element with four-bit data path

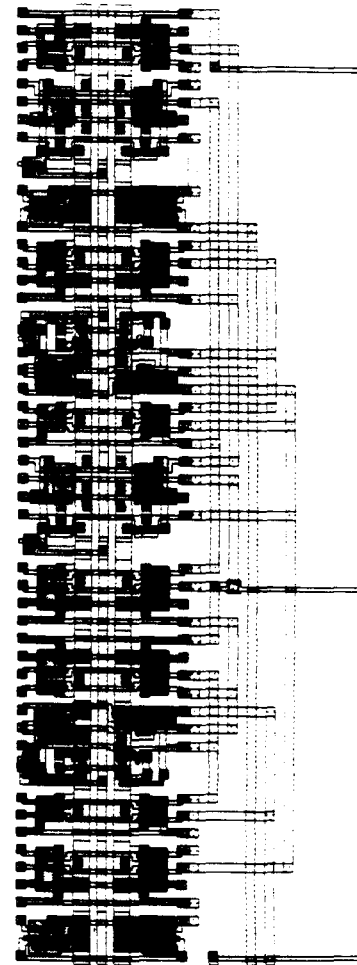


Figure 8: Layout of the control part